Mica

A Modal Interface Compositional Analysis Library

User's guide

Benoît Caillaud

benoit.caillaud@inria.fr

INRIA Rennes – Bretagne Atlantique
Campus de Beaulieu
35042 Rennes cedex
France

version 0.08a

April 10, 2014

Introduction

Mica is an Ocaml library implementing the Modal Interface algebra published in the following paper:

> Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, Roberto Passerone: A Modal Interface Theory for Component-based Design. Fundam. Inform. 108(1-2): 119-149 (2011).

The purpose of Modal Interfaces is to provide a formal support to contract based design methods in the field of system engineering. Modal Interfaces enable compositional reasoning methods on I/O reactive systems.

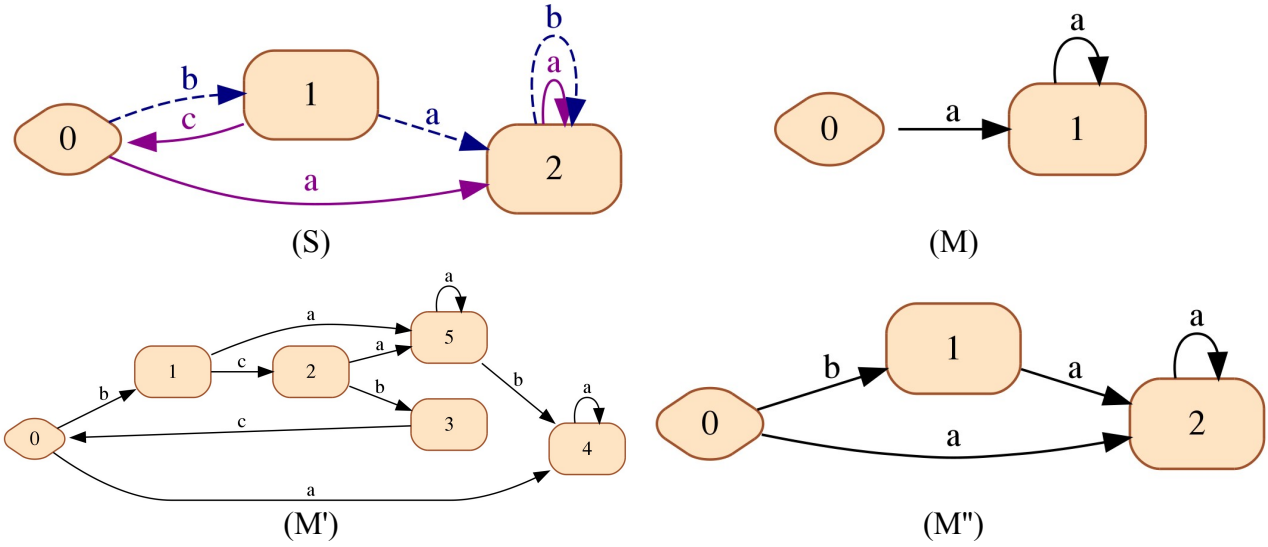
In Mica, systems and interfaces are represented by extension. However, a careful design of the state and event heap enables the definition, composition and analysis of reasonably large systems and interfaces (~10^6 states). The heap stores states and events in a hash table and ensures structural equality (there is no duplication). Therefore complex data-structures for states and events induce a very low overhead, as checking equality is done in constant time.

Thanks to the `Inter` module and the `mica` interactive environment, users can define complex systems and interfaces using Ocaml syntax. It is even possible to define parameterized components as Ocaml functions.

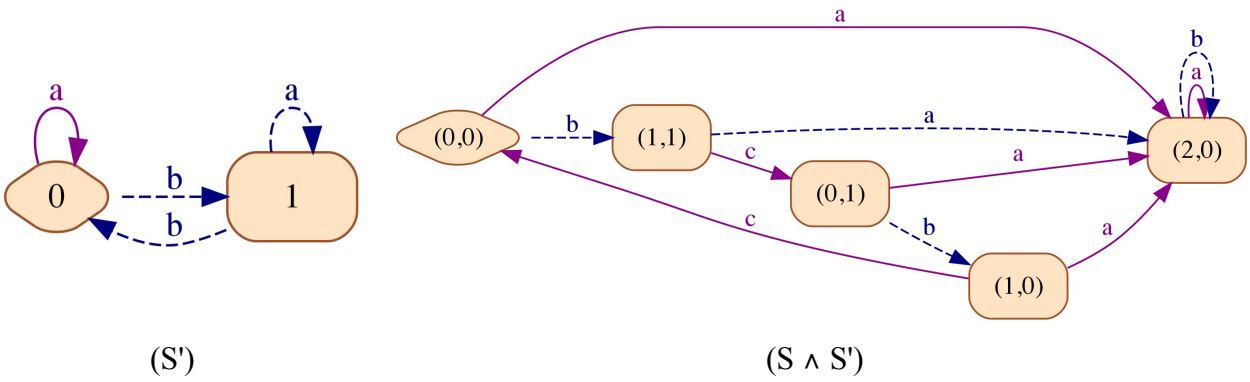
Mica is available as an open-source distribution, under the CeCILL-C Free Software License Agreement, version 1. Users agree to comply with the terms of the license contained in the following document: http://www.cecill.info/licences/Licence_CeCILL-C_V1-en.html

Modal interfaces

Modal interfaces are automata-theoretic specifications with input and output events that can be used capture imprecise or incomplete requirements on the behavior of a system under design. They are deterministic transition systems, with two kinds of transitions: must and may. Must transitions are mandatory, meaning that every correct implementation of the specification is compelled to realize them, while may transitions are optional. For instance, systems (M) and (M') are correct implementations of modal interface (S) given below. In these figures, states in the shape of a diamond are initial. On the contrary, system (M'') is not an implementation of (S). The reason is that after performing event b, the system refuses to perform event c, while in the interface, this event has modality must. Remark that (M') is more involved than (M) since the decision to realize a may transition or not is history dependent.



Modal interfaces can be combined in many different ways, thanks to the conjunction, product and quotient operators. For instance, the figure below details the conjunction of (S) and (S'), where (S') is an interface expressing the property that after an even number of occurrences of event b, event a is compulsory.



Two structural composition operators are available in Mica: The product operator ⊗ reflects the parallel composition of systems, at the level of interfaces. The optimistic parallel composition takes care of input/output incompatibilities. In essence, it computes the least assumption under which every event sent by one component can be received by the other.

Mica also implements two residuation operators: The quotient operator (/) is the adjoint of the product ($S_1 \leq S_3 / S_2$ iff $S_1 \otimes S_2 \leq S_3$), and the weak implication (÷) is a weak adjoint of the conjunction, meaning that $S_1 \leq S_3 \div S_2$ implies $S_1 \wedge S_2 \leq S_3$. However, the converse implication does not hold in general. Interfaces can be compared with the refinement preorder relation. Several generic properties can be checked on interfaces: consistency , completeness and triviality.

Downloading, compiling and installing Mica

This software can be download at: http://www.irisa.fr/s4/download/mica/

The distribution contains several folders: `src` (source code), `doc` (documentation) and `examples` (simple use-cases). It compiles, runs and has been tested on the following platforms:

- Linux,
- MacOS X version 10.4 or later.

It should compile and run on the following platforms. However, this has not been tested:

- Solaris, OpenBSD, FreeBSD,
- Windows NT/2000/XP/Vista/7 with Cygwin.

This software requires the following tools:

- The GraphViz graph mapping toolbox (dot command),
- gmake (only for Darwin and most likely Solaris),
- Ocaml (version 3.12 or later, compiles on earlier versions but has not been thoroughly tested).

To compile this software, change to folder `src` and proceed as follows:

1. Copy file `Makefile_config_generic` to `Makefile_config`
2. Edit file `Makefile_config` and set the installation folders,
3. Run the following `make` or `gmake` command:

```
> make all all.opt doc
```

4. This produces the following libraries and interactive commands:
   - `mica.cma` (Ocaml bytecode library),
   - `mica.cmxa` and `mica.a` (Native codee library),
   - `ocaml_mica` (Ocaml toplevel linked with the mica library),
   - `mica` (the interactive environment).
5. Installation is done as follows:

```
> make install
```

   or as follows, if superuser/administrator privileges are required:

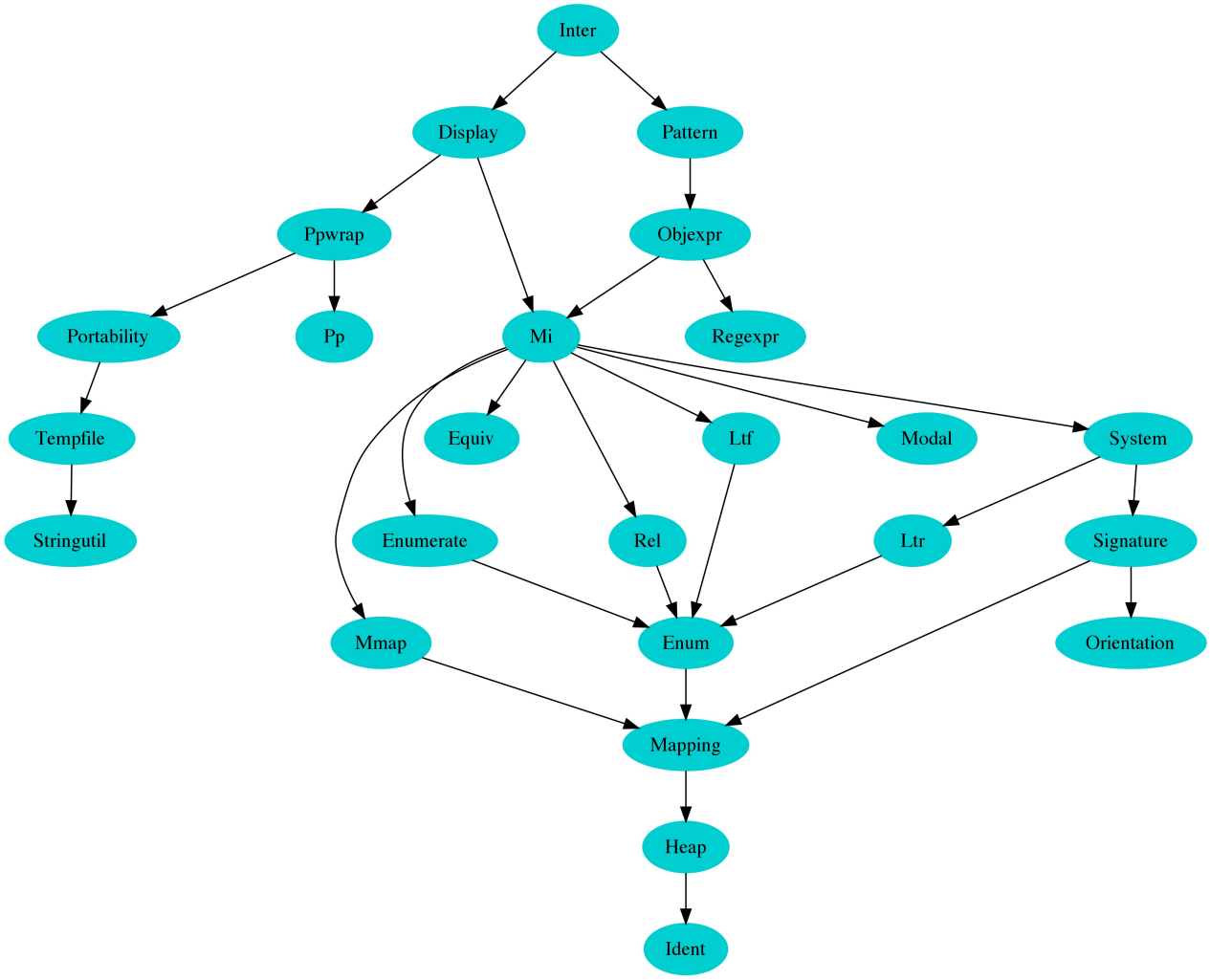
```
> sudo make install
```

The Mica library consists in several modules. Modules that can be used directly are listed below.

Other modules are technical and are not covered in this manual.

Inter	Mica interactive environment commands and functions
Display	Displays systems and interfaces using GraphViz
Pattern	Library of modal patterns using regular expressions
Objexpr	Regular expressions on heap objects and mapping to modal interfaces
Regexpr	Polymorphic regular expressions
Mi	Modal interface algebra
System	System algebra
Signature	Signature algebra
Modal	Algebra of modalities
Orientation	Algebra of I/O orientations
Equiv	Equivalence relations
Rel	Binary relations on heap objects
Ltr	Labelled transition relations
Ltf	Labelled transition functions
Enum	Sets defined by extension
Mmap	Curryfied mappings from pairs of heap objects to any type
Mapping	Mappings from heap objects to any type
Heap	Heap of objects with hashing and structural equality

Their dependency partial order relation is as follows:



The Mica interactive environment

For a detailed documentation of the `mica` commands and functions, see the description of the `Inter` module (below, but also in file `inter.mli` in the `src` folder).

The Inter module

Module Inter defines the commands available in the mica interactive enviroment. Whenever an error occurs at runtime, an exception `Error` is raised. The parameter is an error message.

```
(* Exception [Error _] is raised whenever an error occurs *)

exception Error of string
```

In a nutshell, the tool computes on five types of objects:

- type `value` is used to define states and events;
- type `signature` is used to define I/O alphabets;

- type `system` is used to define systems (component implementations). Systems are parameterized by a signature;
- type `interface` is used to define modal interfaces (component contract, abstraction or behavioral type). Interfaces are parameterized by a signature.
- type `expression` is used to define modal regular expressions, a convenient way to define interfaces.

```
(* type [value]: heap elements *)

type value

(* type [signature]: system and interface I/O signatures *)

type signature

(* type [system]: systems (component implementations) *)

type system

(*type [interface]: interfaces (component behavioral types) *)

type interface

(* type [expression]: modal regular expressions on events *)

type expression
```

Values and regular expressions are allocated in a heap. A hash table allows to retrieve values very quickly and the heap is organized so that every value or regular expression has a unique representation. Thanks to this data structure, equality can be checked in constant time, a very important feature when computing on large state spaces. The following function can be used to initialize or reset the heap:

```
(* [new_session ()] initializes the heap *)

val new_session : unit -> unit
```

Values can be identifiers, characters, strings, booleans, integers and tuples and sets of values. The following functions can be used to construct values in the heap:

```
(* Value factory *)

val ident : string -> value

val char : char -> value

val string : string -> value

val bool : bool -> value

val int : int -> value

val tuple : value array -> value

val set : value list -> value
```

Regular expressions are used to define modal patterns, a library of generic modal interfaces. The following functions can be used to define regular expressions and turn them into modal interfaces:

```
(* Regular expression factory *)

val empty : unit -> expression

val epsilon : unit -> expression

val prefix : value -> expression -> expression

val concat : expression -> expression -> expression

val sum :expression -> expression -> expression

val star : expression -> expression

val shall : value -> expression

val shallnot : value -> expression

(* Generates an interface from an expression *)

val interface_of_expression : signature -> expression -> interface
```

The following functions can be used to print values, signatures, systems and interfaces:

```
(* printers for the above types *)

val print_value : value -> unit

val print_signature : signature -> unit

val print_system : system -> unit

val print_interface : interface -> unit

val print_expression : expression -> unit
```

Statistics about the heap or a given interface are printed by the following functions:

```
(* [print_statistics ()] prints statistics about the heap *)

val print_statistics : unit -> unit

(* Print informations about interfaces and systems *)

val info_interface : interface -> unit
```

Systems and interfaces can be drawn with the following functions. The graphs can be saved as `.dot` GraphViz files. It is also possible to retrieve in the `/tmp` directory a `.svg` or `.pdf` file, depending on the platform operating system.

```
(* Display systems and interfaces *)

val display_system : system -> string (* graph identifier *) -> unit

val display_interface : interface -> string (* graph identifier *) -> unit

(* Generates dot files from systems and interfaces *)
```

```
val dot_file_of_system : system -> string (* graph identifier *) -> string (*
file name *) -> unit

val dot_file_of_interface :
  interface -> string (* graph identifier *) -> string (* file name *) -> unit
```

Signature are defined in a `new_signature ();` … `define_signature ()` block. Functions output and input are used to add an event to the current signature. These two functions can only be called within a `new_signature ();` … `define_signature ()` block.

```
(* SIGNATURES *)

(* [new_signature ()] starts the definition of a signature *)

val new_signature : unit -> unit

(* [define_signature ()] terminates the definition of a signature *)

val define_signature : unit -> signature

(* [output e] defines value [e] to be an output in the current signature *)

val output : value -> unit

(* [input e] defines value [e] to be an input in the current signature *)

val input : value -> unit
```

Systems and interfaces on signature `g` are defined within `new_system g;` … `define_system ()` and `new_interface g;` … `define_interface()` blocks. Function `define_interface` returns a reduced interface.

```
(* SYSTEMS AND INTERFACES *)

(* [new_system g] starts the definition of a system of signature [g] *)

val new_system : signature -> unit

(* [define_system ()] terminates the definition of a system *)

val define_system : unit -> system

(* [new_interface g] starts the definition of an interface with signature [g] *)

val new_interface : signature -> unit

(* [define_interface ()] terminates the definition of an interface *)

val define_interface : unit -> interface
```

Function `void` is used to define the inconsistent interface. Remark that the empty `new_interface g; define_interface()` block also defines the inconsistent interface.

```
(* [void ()] forces the current interface to be inconsistent *)

val void : unit -> unit
```

Function `init` can be used to define the initial state(s) of systems and interfaces. Systems should

have at least one initial states. Interfaces can have zero or one initial state.

```
(* [init q] defines value [q] as initial state of the current system or
interface *)

val init : value -> unit
```

The transition relation of a system is defined by enumerating its transitions. Each transition is defined by defined with the `trans` function:

```
(* [trans q e q'] defines a transition ([q],[e],[q']) in the current system *)

val trans : value -> value -> value -> unit
```

For interfaces, modal transitions are defined with the four following functions. Cannot is the default modality.

```
(* [may q e q'] defines a may transition ([q],[e],[q']) *)

val may : value -> value -> value -> unit

(* [must q e q'] defines a may transition ([q],[e],[q']) *)

val must : value -> value -> value -> unit

(* [inconsistent q e] sets event [e] to be inconsistent in state [q] *)

val inconsistent : value -> value -> unit

(* [cannot q e] sets event [q] to cannot in state [q] *)

val cannot : value -> value -> unit
```

System's and Interface's signatures can be retrieved with the following functions:

```
(* [signature_of_system m] returns the signature of system [m] *)

val signature_of_system : system -> signature

(* [signature_of_interface s] returns the signature of interface [s] *)

val signature_of_interface : interface -> signature
```

Systems can be mapped to interfaces and vice-versa:

```
(* Mappings between interfaces and systems *)

(* [interface_of_system m] maps deterministic system [m] to a rigid interface *)
(* Raises [Error _] if [m] is not deterministic *)

val interface_of_system : system -> interface

(* Extremal implementations. Raises [Error _] if interface is inconsistent *)

val min_implementation : interface -> system

val max_implementation : interface -> system
```

The parallel composition of systems can be computed with the following function. The parallel

composition of I/O-incompatible systems (whenever a system is ready to send an event, while its peer system is not ready to receive this event) produces warnings.

```
(* Composition operator on systems *)

(* [parallel m1 m2] computes the parallel composition of [m1] and [m2] *)

val parallel : system -> system -> system
```

Composition operators on interfaces. Function wimply is the weak implication operator introduced in: G. Goessler and J.-B. Raclet. Modal Contracts for Component-based Design. SEFM'09. The optimistic parallel composition produces warnings whenever incompatible state pairs are reachable in the product interface.

```
(* Composition operators on interfaces *)

(* [conjunction s1 s2] computes the conjunction of interfaces [s1] and [s2] *)

val conjunction : interface -> interface -> interface

(* [product s1 s2] computes the product of interfaces [s1] and [s2] *)

val product : interface -> interface -> interface

(* [quotient s1 s2] computes the residuation of [s1] by [s2] *)

val quotient : interface -> interface -> interface

(*
 *
 * [compatible_quotient a b] computes the compatible residuation of interface
 * [a] by interface [b]. It is the largest [x] such that [x] is compatible with
 * [b] and [product x b] refines [a].
 *
 *)

val compatible_quotient : interface -> interface -> interface

(* [wimply s1 s2] computes the weak implication of [s1] by [s2] *)

val wimply : interface -> interface -> interface

(* [contract g a] computes contract ([g]x[a])/[a] *)

val contract : interface -> interface -> interface

(*
 *
 * [parallel_optimistic s1 s2] computes the optimistic parallel composition
 * of interfaces [s1] and [s2].
 *
 *)

val parallel_optimistic : interface -> interface -> interface
```

Satisfaction, refinement and consistency can be checked with the following functions. Violation of satisfaction or refinement produces warning messages explaining why the relation does not hold.

```
(* Relations on systems and interfaces *)
```

```
(* [satisfies m s] decides whether system [m] satisfies interface [s] *)

val satisfies : system -> interface -> bool

(* [refines s1 s2] decides whether interface [s1] refines interface [s2] *)

val refines : interface -> interface -> bool

(* [is_consistent s] decides whether interface [s] is consistent *)

val is_consistent : interface -> bool

(* [is_complete s] checks whether interface [s] is complete *)

val is_complete : interface -> bool

(* [is_trivial s] checks whether interface [s] is trivial *)

val is_trivial : interface -> bool
```

Interfaces can be made minimal with the following function. It implements an algorithm based on the computation of the largest congruence of the interface.

```
(* [minimize s] minimizes interface [s] *)

val minimize : interface -> interface
```

The structure of states becomes more and more complex when iterating on composition operators. Although this is not really an issue regarding computational complexity, this can obfuscate the graphical display of a modal interface. The following function simplifies the state structure of a modal interface by computing an injective mapping from the set of reachable states to integers:

```
(* [simplify s] simplifies the state structure of interface [s] by mapping state
labels to integers *)

val simplify : interface -> interface
```

Whenever an interface needs to be inspected and its state structure is too complex to be represented graphically, projections on a sub-alphabet turns out to be helpful. The projection of C on sub-alphabet L is the least abstraction of C with signature L.

```
(*
 *
 * [abstract l a] computes a projection of interface [a] on sub-alphabet [l].
 * The resulting interface is the least abstraction of [a] in the class of
 * interfaces over alphabet [l].
 *
 *)

val abstract : value list -> interface -> interface

(*
 *
 * [project s a] computes a projection of interface [a] on signature [s]. The
 * resulting interface is the largest projection of [a] that is permissive to
 * inputs and that controls its outputs.
 *
```

```
 *)

val project : signature -> interface -> interface
```

Several modal patterns can be used to help the definition of modal interfaces:

```
(* Patterns *)

(*
 *
 * [every_must g u] produces an interface with signature [g]
 * such that for every occurrence of word [u], the last event
 * of [u] has modality must. Exception [Error _] is raised if
 * [u] is not in signature [g].
 *
 *)

val every_must : signature -> value array -> interface

(*
 *
 * [every_cannot g u] produces an interface with signature [g]
 * such that after every occurrence of word [u], last event excepted,
 * the last event of [u] has modality cannot. Exception
 * [Invalid_argument _] is raised if [u] is not in signature [g].
 *
 *)

val every_cannot : signature -> value array -> interface

(*
 *
 * [after_must g x e] produces an interface with signature [g]
 * such that after the occurrence of a word accepted by
 * expression [x], event [e] has modality must.
 * Exception [Error _] is raised if [x] or [e] are not in
 * signature [g].
 *
 *)

val after_must : signature -> expression -> value -> interface

(*
 *
 * [after_cannot g x e] produces an interface with signature [g]
 * such that after the occurrence of a word accepted by
 * expression [x], event [e] has modality cannot.
 * Exception [Error _] is raised if [x] or [e] are not in
 * signature [g].
 *
 *)

val after_cannot : signature -> expression -> value -> interface
```

Maximal consistent subsets of a set of interface can be computed with the following function:

```
(*
 *
 * [max_consistent_sets a] computes the list of maximal consistent subsets of
 * [a]. [a] is an array of pairs identifier * interface.
```

```
 *
 *)

val max_consistent_sets : ('a * interface) array -> 'a list list
```

Example

Examples can be found in folder `examples`. Follow instructions in file `readme.txt` located there. The parking use-case is detailed below. Shell and Mica command line input is colored blue, while tool output is colored yellow.

Start the Mica interactive environment:

```
> mica
        Objective Caml version 3.12.1

 ************************************************************************
 *                                                                      *
 *        Mica : A Modal Interface Compositional Analysis Library       *
 *                                                                      *
 *                      Benoit Caillaud, INRIA-Rennes                   *
 *                       <Benoit.Caillaud@inria.fr>                     *
 *                                                                      *
 * Copyright Benoit Caillaud, Institut National de Recherche en         *
 * Informatique et  Automatique, September 2011.                        *
 *                                                                      *
 * Distributed under the CeCILL-C Free Software Licence Agreement:      *
 * http://www.cecill.info/licences/Licence_CeCILL-C_V1-en.html          *
 *                                                                      *
 ************************************************************************

$Id: inter.ml 374 2011-10-05 11:47:23Z bcaillau $

Welcome to Mica, version 0.08a
#
```

Define an array of states:

```
# let q = [| int 0; int 1; int 2; int 3; int 4; int 5; int 6; int 7 |];;
val q : Inter.value array =
    [|<abstr>;  <abstr>;  <abstr>;  <abstr>;  <abstr>;  <abstr>;  <abstr>;
<abstr>|]
```

Define events. Notice event `vehicule_sense` takes a boolean parameter:

```
# let vs b = tuple [| ident "vehicule_sense"; bool b |];;
val vs : bool -> Inter.value = <fun>
# let tr = ident "ticket_request";;
val tr : Inter.value = <abstr>
# let ti = ident "ticket_issue";;
val ti : Inter.value = <abstr>
# let td = ident "ticket_discard";;
val td : Inter.value = <abstr>
# let tg = ident "ticket_grasp";;
val tg : Inter.value = <abstr>
# let go = ident "gate_open";;
val go : Inter.value = <abstr>
# let gc = ident "gate_close";;
val gc : Inter.value = <abstr>
```

Define a signature consisting of two input events:

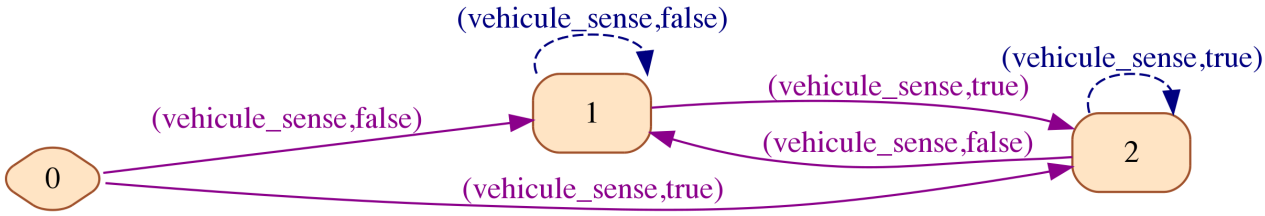
```
# let sigma_entry_gate_receptive =
  new_signature ();
  input (vs true);
  input (vs false);
  define_signature ();;
val sigma_entry_gate_receptive : Inter.signature = <abstr>
```

Define an interface on the signature defined above.

```
# let itf_entry_gate_receptive =
  new_interface sigma_entry_gate_receptive;
  init q.(0);
  must q.(0) (vs false) q.(1);
  must q.(0) (vs true) q.(2);
  may q.(1) (vs false) q.(1);
  must q.(1) (vs true) q.(2);
  must q.(2) (vs false) q.(1);
  may q.(2) (vs true) q.(2);
  define_interface ();;
val itf_entry_gate_receptive : Inter.interface = <abstr>
```

Display the interface:

```
# display_interface itf_entry_gate_receptive "itf_entry_gate_receptive";;
- : unit = ()
```



Define and display two other signatures and interfaces:

```
# let sigma_entry_gate_request =
  new_signature ();
  input (vs true);
  input (vs false);
  input tr;
  define_signature ();;
val sigma_entry_gate_request : Inter.signature = <abstr>
# let itf_entry_gate_request =
  new_interface sigma_entry_gate_request;
  init q.(0);
  must q.(0) tr q.(0);
  may q.(0) (vs false) q.(1);
  may q.(0) (vs true) q.(2);
  may q.(1) (vs false) q.(1);
  may q.(1) (vs true) q.(2);
  may q.(1) tr q.(1);
  may q.(2) (vs false) q.(1);
  may q.(2) (vs true) q.(2);
  must q.(2) tr q.(3);
  may q.(3) (vs false) q.(1);
  may q.(3) (vs true) q.(3);
  may q.(3) tr q.(3);
  define_interface ();;
val itf_entry_gate_request : Inter.interface = <abstr>
# display_interface itf_entry_gate_request "itf_entry_gate_request";;
```

```
- : unit = ()
# let sigma_entry_gate_ticket =
  new_signature ();
  input (vs true);
  input (vs false);
  input tr;
  output ti;
  output td;
  input tg;
  output go;
  output gc;
  define_signature ();;
val sigma_entry_gate_ticket : Inter.signature = <abstr>
# let itf_entry_gate_ticket =
  new_interface sigma_entry_gate_ticket;
  init q.(0);
  may q.(0) (vs false) q.(0);
  may q.(0) (vs true) q.(1);
  may q.(0) tr q.(0);
  may q.(1) (vs false) q.(0);
  may q.(1) tr q.(2);
  may q.(2) (vs false) q.(0);
  may q.(2) tr q.(2);
  must q.(2) ti q.(3);
  may q.(3) (vs false) q.(4);
  must q.(3) tg q.(5);
  may q.(3) tr q.(3);
  must q.(4) td q.(0);
  may q.(4) (vs true) q.(3);
  may q.(4) tr q.(4);
  may q.(5) (vs false) q.(0);
  must q.(5) go q.(6);
  may q.(5) tr q.(5);
  may q.(6) (vs false) q.(7);
  may q.(6) tr q.(6);
  must q.(7) gc q.(0);
  may q.(7) (vs true) q.(6);
  may q.(7) tr q.(7);
  define_interface ();;
val itf_entry_gate_ticket : Inter.interface = <abstr>
# display_interface itf_entry_gate_ticket "itf_entry_gate_ticket";;
- : unit = ()
```

Compute the conjunction of the three interfaces:

```
# let itf_entry_gate =
  conjunction
    (conjunction
       itf_entry_gate_receptive
       itf_entry_gate_request)
    itf_entry_gate_ticket;;
val itf_entry_gate : Inter.interface = <abstr>
```

Checks whether the conjunction is consistent:

```
# is_consistent itf_entry_gate;;
- : bool = true
```

Displays the conjunction (figure below):

```
# display_interface itf_entry_gate "itf_entry_gate";;
- : unit = ()
```

Prints heap statistics:

```
# print_statistics ();;
Heap      statistics:      table_length=79397,      number_of_entries=36,
sum_of_bucket_lengths=91,  minimal_bucket_length=0,  mean_bucket_length=0,
maximal_bucket_length=7
-      : unit = ()
```